Systèmes concurrents

Philippe Quéinnec

ENSEEIHT
Département Sciences du Numérique

25 septembre 2020

Quatrième partie

L'interblocage

Contenu de cette partie

- Notion d'interblocage
- Caractérisation des situations d'interblocage
- Protocoles de traitement de l'interblocage
  - préventifs
  - curatifs
- Apport déterminant d'une bonne modélisation/formalisation pour la recherche de solutions

Plan

Allocation de ressources multiples

- Ressources banalisées, réutilisables, regroupées en classes
- Une activité demande un certain nombre de ressources dans chaque classe
- Demande bloquante, ressources allouées par un gérant de ressources
- Interface du gérant :
  - `demander` : (IdClasse → natural) → (Set of IdRessource)
  - `libérer` : (Set of IdRessource) → unit
- Le gérant :
  - rend la ressource réutilisable, lors d'une libération ;
  - libère les ressources détenues, à la terminaison d'une activité.

Exemples

Exclusion mutuelle

1 classe, 1 ressource, demande $= 1$

Sémaphore

1 classe, R ressources, demande $= 1$

Philosophes

N classes de 1 ressource, P_i demande R_i et $R_{i \oplus 1}$

Allocateur mono-classe

1 classe de R ressources, demande $\in [1..R]$

Lecteurs/rédacteurs

1 classe de N ressources, demande $= 1$ ou $= N$

Propriétés temporelles

Exprimer la correction

Pour *toutes* les exécutions possibles,
à *tout* moment d'une exécution.

- sûreté : rien de mauvais ne se produit
  l'exclusion mutuelle, les invariants d'un programme
- vivacité : quelque chose de bon finit par se produire
  l'équité, l'absence de famine, la terminaison d'une boucle
- (possibilité : pour *certaines* exécutions, . . .)

La vivacité

- Progression : si *des* activités déposent des requêtes de manière continue, *une* des requêtes finira par être satisfaite ;
- Vivacité faible : si une activité dépose sa requête de manière continue, elle finira par être satisfaite ;
- Vivacité forte : si une activité dépose une requête infiniment souvent, elle finira par être satisfaite ;
- Vivacité FIFO : si une activité dépose une requête, elle sera satisfaite avant tout autre requête (conflictuelle) déposée ultérieurement.

Famine

Une activité est en famine lorsqu'elle attend infiniment longtemps la satisfaction de sa requête (elle n'est jamais satisfaite).

(les activités sont supposées se comporter « correctement »)

L'interblocage

Allocation de ressources réutilisables

- non réquisitionnables
- non partageables
- en quantités entières et finies
- dont l'usage est indépendant de l'ordre d'allocation

Problème : A_1 demande R_1 puis demande R_2,
A_2 demande R_2 puis demande R_1
entrelacement $\rightarrow$ risque d'interblocage.

1. A_1 demande et obtient R_1
2. A_2 demande et obtient R_2
3. A_1 demande R_2 et se bloque
4. A_2 demande R_1 et se bloque

Définition de l'interblocage

Interblocage : définition

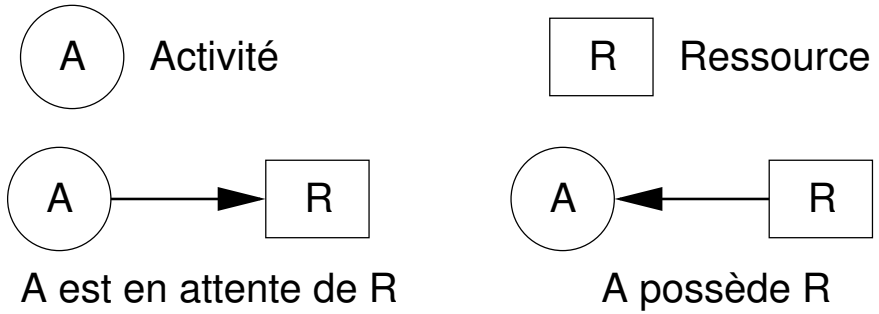
Un ensemble d'activités est en *interblocage* (*deadlock*) si et seulement si toute activité de l'ensemble est en attente d'une ressource qui ne peut être libérée que par une autre activité de l'ensemble.

Pour l'ensemble d'activités interbloquées, interblocage $\equiv$ négation de la progression

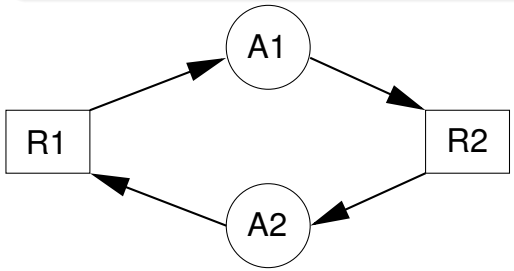
absence de famine $\rightarrow$ absence d'interblocage

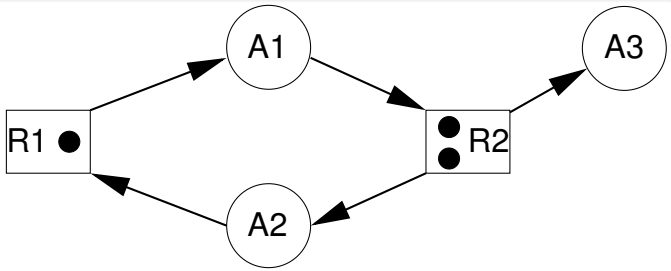
L'interblocage est un état stable.

Graphe d'allocation



Interblocage ≡ cycle/knot dans le graphe d'allocation





Solutions

- Prévention : empêcher la formation de cycles
- Détection + guérison : détecter l'interblocage, et l'éliminer

Plan

Éviter le blocage

Pas de blocage $\rightarrow$ pas d'arc sortant $\rightarrow$ pas de cycle !

Éviter l'accès exclusif

Ressource virtuelle : imprimante, fichier

Éviter la redemande bloquante

Acquisition non bloquante : le demandeur peut ajuster sa demande si elle ne peut être immédiatement satisfaite

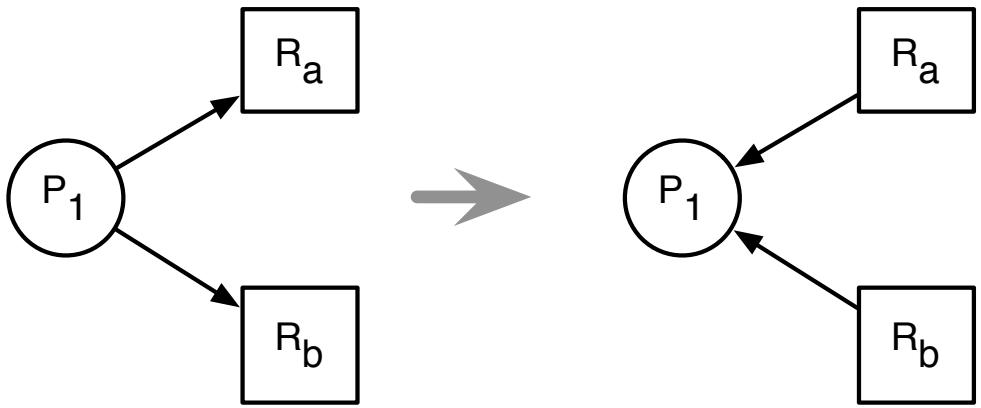
Éviter les demandes fractionnées

Demande unique

Allocation globale : chaque activité demande et obtient en bloc, en une seule fois, toutes les ressources nécessaires

→ une seule demande pour chaque activité

- demande satisfaite → arcs entrants uniquement
- demande non satisfaite → arcs sortants (attente) uniquement



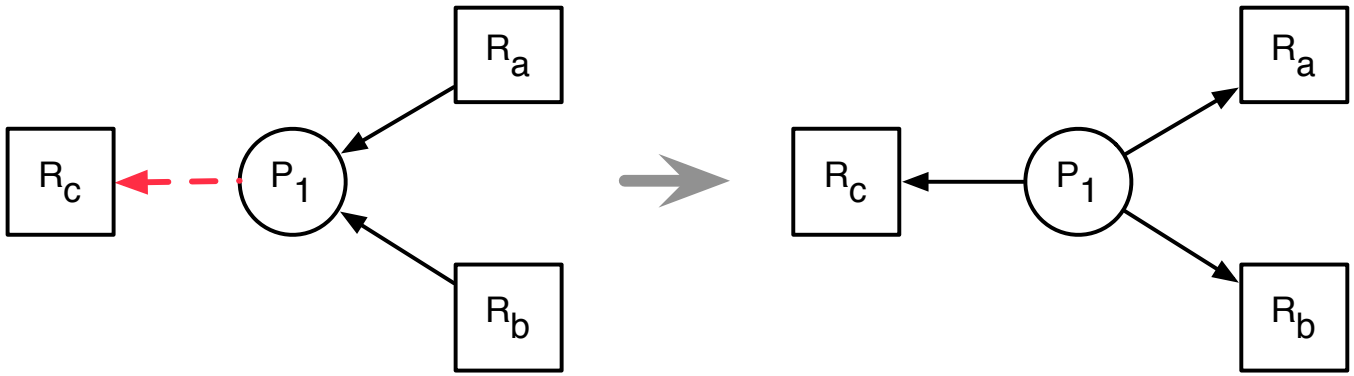
– connaissance a priori des ressources nécessaires
– sur-allocation et risque de famine

Réquisition des ressources allouées

Permettre la réquisition des ressources allouées

Inverser les arcs entrants d'une activité si création d'arcs sortants.
Une activité demandeuse doit :

- libérer les ressources qu'elle a obtenues
- réobtenir les ressources libérées, avant de pouvoir poursuivre
  → risque de famine

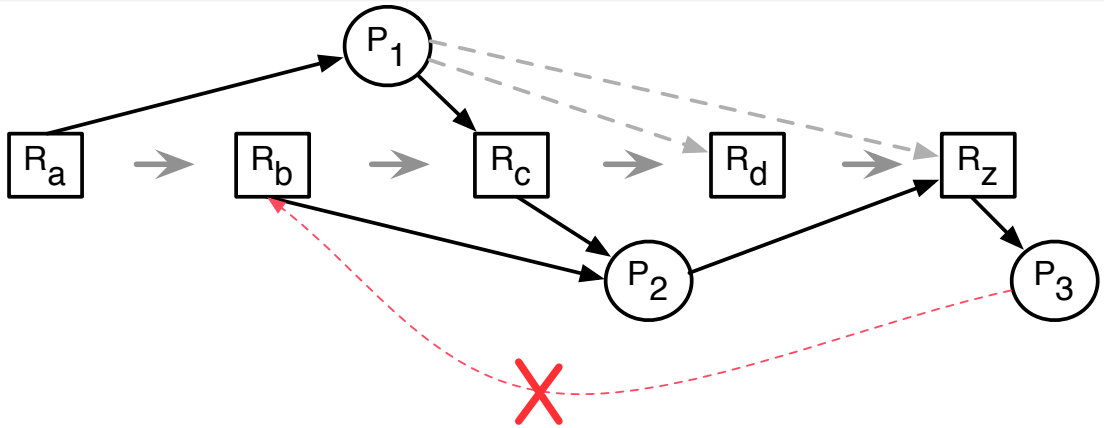


(optimisation : restitution paresseuse des ressources : libération que si la demande est bloquante)

Éviter l'attente circulaire

Classes ordonnées

- Un ordre est défini sur les classes de ressources
- Toute activité doit demander les ressources selon cet ordre

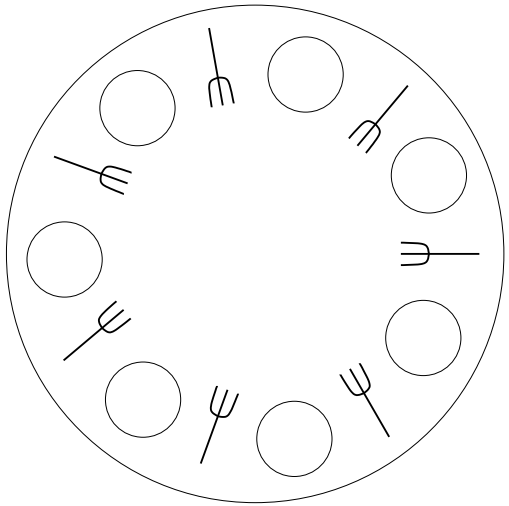


Pour chaque activité, les chemins du graphe d'allocation vont des ressources inférieures (déjà obtenues) aux supérieures (demandées) $\rightarrow$ absence de cycle

Principale solution pour éviter l'interblocage dû à des verrous multiples.

Exemple : Philosophes et spaghettis – Dijkstra

N philosophes sont autour d'une table.
Il y a une assiette par philosophe, et
une fourchette entre chaque assiette.
Pour manger, un philosophe doit
utiliser les deux fourchettes adjacentes
à son assiette (et celles-là seulement).



Un philosophe peut être :

- penseur : il n'utilise pas de fourchettes ;
- mangeur : il utilise les deux fourchettes adjacentes ; aucun de ses voisins ne peut manger ;
- demandeur : il souhaite manger mais ne dispose pas des deux fourchettes.

Ce problème est analogue au problème de l'allocateur multiclasse multiressource.

Exemple : philosophes et interblocage

Risque d'interblocage

Chaque philosophe demande sa fourchette gauche et l'obtient. Puis quand tous ont leur fourchette gauche, chaque philosophe demande sa fourchette droite et se bloque. $\Rightarrow$ interblocage

Solutions

Allocation globale : chaque philosophe demande simultanément les deux fourchettes.

Non conservation : quand un philosophe essaye de prendre sa seconde fourchette et qu'elle est déjà prise, il relâche la première et se met en attente sur la seconde.

Classes ordonnées : imposer un ordre sur les fourchettes $\equiv$ tous les philosophes prennent d'abord la gauche puis la droite, sauf un qui prend d'abord droite puis gauche.

Esquive

Avant toute allocation, évaluation du risque (futur) d'interblocage.

L'algorithme du banquier

- Chaque activité annonce le nombre maximal de ressources qu'elle demandera.
- L'algorithme maintient le système dans un état fiable, i.e. tel qu'il existe toujours une possibilité d'éviter l'interblocage dans le pire des scénarios.
- En cas de danger détecté, le requête est mise en attente (comme si les ressources n'étaient pas disponibles).
- Un état est fiable s'il existe une séquence d'allocations satisfaites qui permet à tous les processus d'obtenir toutes leurs ressources

12 ressources, $A_0/A_1/A_2$ annoncent $10/4/9$

	max	poss.	demande	
A_0	10	5		
A_1	4	2	$+1$	oui
				$(5+4+2 \leq 12) \wedge (10+0+2 \leq 12) \wedge (0+0+9 \leq 12)$
A_2	9	2	$+1$	non, bloque
				$(10+2+3 > 12)$
				ou $((5+4+3 \leq 12) \wedge (10+0+3 > 12) \wedge (5+0+9 > 12))$
				ou $(5+2+9 > 12)$

Algorithme du banquier (1/2)

fonction allouer(id, demande)

```
var  disponibles : entier = N
     annoncées, allouées : tableau [1..NbProc] de entier

tant que demande non satisfaite faire
  si étatFiable({1..NbProc}, disponibles - demande) alors
      allouées[id] ← allouées[id] + demande
      disponibles ← disponibles - demande
  sinon <bloquer l'activité>
  finsi
fintq
```

fonction terminer(id)

```
disponibles ← disponibles + allouées[id]
allouées[id] ← 0
<débloquer (intelligemment) toutes les bloquées>
```

Algorithme du banquier (2/2)

```
fonction étatFiable(demandeurs:ensemble de 1..NbProc,
                    dispo: entier): booléen
var vus, S : ensemble de 1..NbProc = ∅, ∅;
    ok : booléen = (demandeurs = ∅);
début
  répéter
    S ← {p ∈ demandeurs \ vus :
                  annoncées[p]-allouées[p] ≤ dispo}
    si S ≠ ∅ alors
       choisir d ∈ S
       vus ← vus ∪ {d}
       ok ← étatFiable(demandeurs-{d},
                       dispo+annoncées[d]-allouées[d])
    finsi
  jusqu'à (S = ∅) ou (ok)
  renvoyer ok
fin
```

Plan

Détection

- Construire le graphe d'allocation
- Détecter l'existence d'un cycle (ressources uniques) ou d'un « knot » (ressources multiples équivalentes)

  knot = composante fortement connexe terminale
  = ensemble S d'activités/ressources tels que
  $\forall s \in S :$ successeur$(s) \in S$
  $\wedge\ s \in$ activité $\Rightarrow$ successeur$(s) \neq \emptyset$
  $\wedge\ s \in$ ressources $\Rightarrow$ card(succ(s)) = nb ress.
- Algorithme coûteux $\rightarrow$ périodiquement et non à chaque allocation (l'interblocage est un état stable $\rightarrow$ il finira par être détecté)

Guérison

Réquisition des ressources détenues par une (ou plusieurs) activité(s).

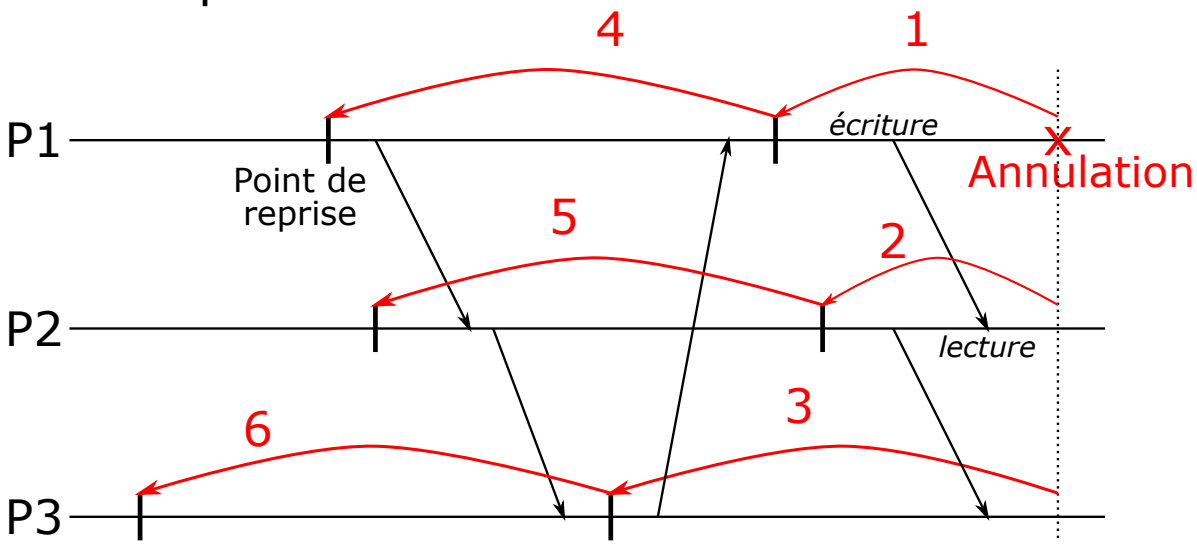
- Comment choisir le(s) activités victimes (la dernière qui a alloué, la plus grosse/petite consommatrice, notion d'importance...) ?
- Annulation de l'activité ou retour en arrière ?
- Solution coûteuse (détection + choix + travail perdu), pas toujours acceptable (systèmes interactifs, systèmes embarqués).
- Simplifie l'allocation.
- Points de reprise pour retour arrière.

Points de reprise

Définition

Sauvegarde d'états intermédiaires pour éviter de perdre tout le travail.

Utilisé pour les transactions en base de données
Effet domino : l'annulation d'une action entraîne l'annulation d'une deuxième action qui. . .



Plan

Conclusion

- Usuellement : interblocage = inconvénient occasionnel
  - $\rightarrow$ laissé à la charge de l'utilisateur / du programmeur
  - utilisation de méthodes de prévention simples (p.e. classes ordonnées)
  - ou détection empirique (délai de garde) et guérison par choix manuel des victimes
- Cas particuliers : systèmes ouverts contraints par le temps
  - systèmes interactifs, systèmes embarqués
  - recherche de méthodes efficaces, prédictibles, ou automatiques
  - compromis à réaliser entre
    - la prévention (statique, coûteuse, restreint le parallélisme)
    - la détection/guérison (moins prédictible, coûteuse quand les conflits sont fréquents)
- Approche sans blocage (cf synchronisation non bloquante)

Autres difficultés de synchronisation

- Accès non protégé à un état partagé (conflits d'écriture, visibilité d'états transitoires). Se manifestent souvent par des comportements non reproductibles (heisenbugs)
  Variables partagées ⇒ verrous
  Outils d'analyse statique pour identifier de potentiels problèmes
- Invalidation d'un invariant (souvent facile à détecter, mais coûteux et non correctible en général)
- Famine : difficilement prouvable, impossible à détecter (sauf stratégie régulière : FIFO)
- « Livelock » : boucle d'états distincts mais où une activité (au moins) ne progresse pas vers sa terminaison (p.e. boucle allocation - détection d'interblocage - préemption - retentative)